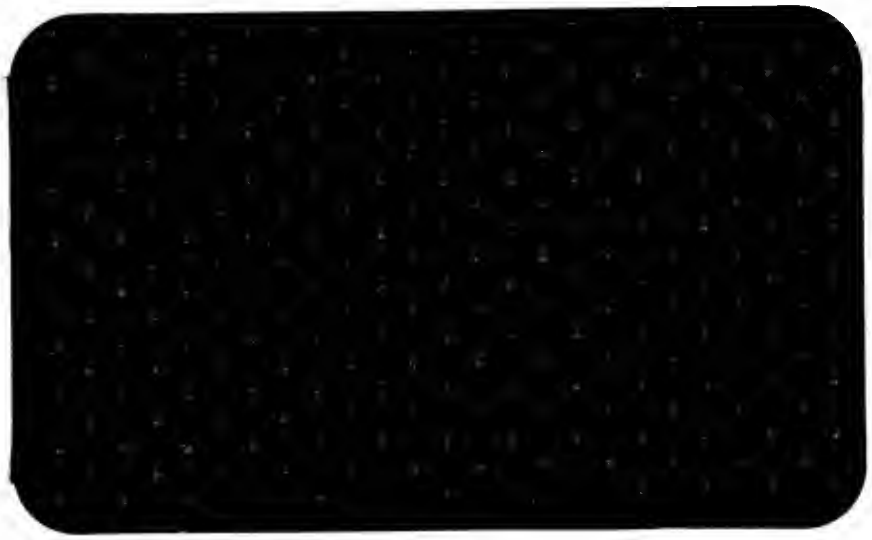
THE CIMS PL/I COMPILER

BY

PAUL W. ABRAHAMS

APRIL 1979

REPORT NO. 013



THE CIMS PL/I COMPILER

BY

PAUL W. ABRAHAMS

APRIL 1979

REPORT NO. 013

The research described in this paper was supported in part by Control Data Corporation Research Grant C08AA, and in part by United States Department of Energy Research Grant EY-76-C-02-3077.

Abstract

CIMS PL/I is an implementation of PL/I on the Control Data 6600 computer. The compiler is itself written almost entirely in PL/I. Since a PL/I compiler is an inherently large object, building one is a good way to test the limits of ideas on compiler construction and programming methodology. The compiler is a multi-pass affair, and the first three passes are almost entirely machine-independent. Two primary kinds of internal representations are used: a sequential form for information with relatively little interreferencing, and a pointer-linked form for other objects such as declarations and block descriptors. The rearmost machine-independent representation is the Second Intermediate Language, IL2, which treats the program as a set of instructions to a virtual PL/I machine. The instructions include both executable instructions and storage-defining instructions. Parsing is done semi-automatically, using a preprocessor to build essential tables. Reasons for not using a fully automatic parser are given. Good error handling is achieved by generalizing the syntactic forms accepted by the parser, and preparing the later passes to cope with these generalized forms. Preprocessors that derive useful information from the source text of the compiler itself are used for various purposes.

1. INTRODUCTION

CIMS PL/I is an implementation of PL/I on the Control Data 6600 computer. The most challenging aspect of implementing PL/I is dealing with the sheer size and complexity of the language; since a PL/I compiler is an inherently large object, building one is a good way to test the limits of ideas on compiler construction and programming methodology. Version 1 of CIMS PL/I has been in active use since 1973 and includes roughly 70% of the full language, but is limited by some severe design flaws. Version 2 is now under development; the parser and the declaration processor are in excellent working order, but the later passes are still being worked on. In this paper I shall describe Version 2 as though it already existed in full. Version 2 is intended to implement full ANSI Standard PL/I [1], and the passes that have been completed accept the full language. Both versions are themselves written almost entirely in PL/I, and have been developed using bootstrapping methods.

2. OVERVIEW OF THE DESIGN

The compiler is a multi-pass affair; each pass performs a transformation from one representation of the program (or of some part of the program) to another representation. We can view the compiler either in terms of the representations or in terms of the passes. The representations are:

1. The original source program, consisting of a sequence of independently-processed external procedures.

2. A sequence of integers and pointers (Intermediate Language 1, or IL1) that corresponds straightforwardly to the content of the executable statements of the procedure, together with a collection of tree structures that represent the hierarchy of blocks, the set of declarations, the set of procedure entry-points, and similar constructs. The IL1 code is segmented by block nesting level as discussed below.

3. A modification of the previous form in which declarations not given by the user have been generated, and every declaration has a complete set of attributes.

4. A sequence of operations (IL2) that represent instructions to a hypothetical PL/I machine. IL2, like all the preceding representations, is machine-independent except for constructs such as OP-

TIONS and ENVIRONMENT for which the syntax and semantics are machine-dependent by definition. IL2 is described in more detail in a following section.

5. A machine-dependent but register-independent, unoptimized representation (IL3).

6. The final code in the form of load modules, one per external procedure.

The passes are:

1. The initializer, which initializes tables and processes the compilation options specified for each external procedure.

2. The parser, which transforms the procedure into IL1 and the tree structures.

3. The declaration processor, which generates declarations not given by the programmer and determines the complete set of attributes for each declaration.

4. Code generator 1 (CG1), which produces IL2.

5. Code generator 2 (CG2), which produces IL3.

6. Code generator 3 (CG3), which produces the load modules.

7. The cross-reference and attribute lister.

8. The error-message processor.

Each of these passes can be a separate overlay, although it is appropriate to combine several consecutive passes into one overlay as long as the length of the longest single-pass overlay is not exceeded. For instance, the initializer and the parser are in one overlay. The error message processor is in a separate overlay because it often is not called at all, and in order to avoid to avoid swelling the overlays with the texts of error messages.

3. INTERNAL REPRESENTATIONS

The choice of internal representations was one of the more subtle decisions to be made in designing the compiler. Different internal representations are used for different kinds of objects; the choice depends primarily on how the objects are generally accessed. The two primary forms are sequences of code items and pointer-linked structures. The code items are either integers or pointers. The sequential form is used for information that is usually examined only once and has little interreferencing, i.e., few references from one code item to another. Most information is kept in sequential form; examples are:

1. Executable code.

2. Parsed representations of extent expressions and initial value lists found in declarations.

3. The operations of IL2, the output of the first code generator.

Information in sequential form is partitioned into segments, which are managed by a software paging system. Each segment consists of a sequence of pages. At any time there is a current input segment and a current output segment (although either of these can be undefined). Operations are provided to add a code item to the current output segment and to read a code item from the current input segment. (Making the segment number an explicit parameter of these operations would imply that the current segment numbers would have to be universally visible.) There are also operations to select an output segment and to select a given position in a given input segment. Reading or writing a single code item is an uncomplicated operation as long as no page boundaries are crossed. If the program is small then all pages can be kept in main store; if not, then some of the pages are written to secondary store. The page and segment tables keep track of where everything is.

The code generated during each pass is grouped into segments so as to sort it into the order needed in the following passes. For instance, the parsing pass uses one segment per static nesting level. Code items generated from the outermost block go into one segment, those from blocks nested one level deep go into a different segment, and so forth. However, an entirely different segment is used for executable information found in declarations, such as extent expressions (e.g., "L+2" in "CHARACTER(L+2)") and initial attributes. In later passes it is possible to process the entire text of a procedure without having to switch context back and forth in order to process a contained procedure. In the output of the first code generation pass (IL2), there are additional segments for storage-defining operations, grouped by storage class, and for the prologue code required for each block.

In Version 1, pointer-linked structures were used throughout. Since the CDC 6600 is not a virtual-memory machine, the space requirements of the compiler were a major problem for its users. The software paging system helps a great deal, since the space needed for compilation depends only slightly on the amount of executable code. However, the space requirement still depends on the amount of declarative information present, and the software paging system is an unpleasantly complicated mechanism. A large virtual memory would change the picture considerably, and the Multics PL/I compiler [2], which does run on a virtual-memory machine, takes advantage of that fact by using pointer-linked structures extensively. Even with sequential code, a virtual memory would very much simplify the organization. This situation is an interesting instance of how machine architecture can have a profound influence on compiler design, quite independently of the object code produced by the compiler.

Declarations, entry-point lists, block descriptors, and similar structures are represented by pointer-linked structures (as in Version 1). These objects are referred to from many different places, and are never removed from main store. Thus the space needed for compilation increases as the number of these structures increases. Pointer-linked structures are used sparingly. For instance, the contents of the INITIAL attribute, which is potentially a very long list, are kept in sequential rather than structure form. Similarly, extent expressions within declarations are represented compactly. If the extent expression is an integer or an asterisk (as it usually is), then the extent expression is represented by the integer value or by an asterisk indicator; otherwise the extent expression is represented by an integer giving the location of the executable code for the expression within the declaration segment.

Tokens are stored uniquely; that is to say, all occurrences of a token such a "ABC" or "82" are represented by a single object, and the token is then designated by a pointer to that object. The token for an identifier contains a pointer to a linked list of declarations of that identifier. Similarly, all occurrences of a given qualified name within a given block are represented by a pointer to a single unique object. Thus all occurrences of "A.B.C" within a given block are represented by the same pointer (and similarly for "A", since a qualified name can have just one identifier). Occurrences of "A.B.C" within a different block are represented by a pointer to a different object, since "A.B.C" might have different meanings in the two blocks. This scheme avoids wasting the space needed for multiple copies of the representation of a qualified name. It also conserves processing time, since the reference resolution process only has to be carried out once per name object. An occurrence of a qualified name within an expression that is part of a declaration needs special treatment, on account of a pathological case that can arise when such an expression is copied into another block by means of a LIKE attribute or a DEFAULT statement.

4. THE SECOND INTERMEDIATE LANGUAGE

The second intermediate language (IL2) represents the program being compiled as a set of instructions for a virtual PL/I machine. Figure 1 gives an example of the IL2 code for a short procedure. The instructions themselves have variable length, and include both executable operations and storage-defining operations. The storage-defining operations create storages, which hold both intermediate results and named variables and are referenced by executable operations. The storage-defining operations are, in effect, instructions to the next pass on how to construct a symbol table.

The layout of the IL2 storages in real storage is left undefined at the IL2 level, so the following code generator must record in its symbol table the actual storage mapping. Although IL2 code is independent of the storage mapping algorithm, any extents required for storage mapping are explicitly calculated by IL2 instructions. For instance, given the declaration

```
DECLARE
   1 S BASED(P(I)),
      2 A(M*N) FIXED BINARY,
      2 B FLOAT BINARY;
```

the instructions that define the storage for S assume the existence of a location containing the upper bound of the array A. When B is referenced, the value of M*N is calculated and placed in that location before the reference to B is attempted. (The location of B depends on the size of A,

```
 TEST:  PROCEDURE;
        DECLARE A FIXED BINARY(18,0);
        DECLARE B FIXED DECIMAL(4,2);
        B = 4*A+6.283;
        END TEST;
```

<u>Constant Segment</u>

```
1000 (1,1)               representation of 4
     RXD/A(1,0)
1001 (2,5)               representation of 6.283
     RXD/A(4,3)
```

<u>Constant String</u>: 46.283TEST()

```
2000 RXB/A(18,0)         storage for A
2001 RXD/A(4,3)          storage for B
```

<u>Prologue Segment</u>

```
     ENTRY(1,10,7,6,0)   entrypoint  1 at label 10,
                         descriptor "TEST()",
                         0 parameters
     PROLOG2
```

<u>Main Code Segment</u>

```
     LABEL(10)
     DS/TR
4000 RXB/A(4,0)          get 4 in binary
     MOVE(4000,1000)
     DS/TR
4001 RXB/A(23,0)         result of 4*A
     MULT(4001,4000,2000) binary multiply
     DS/TR
4002 RXB/A(15,10)        get 6.283 in binary
     MOVE(4002,1001)
     DS/TR
4003 RXB/A(39,10)        4*A+6.283
     PLUS(4003,4001,4002)
     MOVE(2001,4003)     move result to B
     RETURN
     ENDBLOCK
```

Figure 1. An Example of IL2 Code.
(continued on next page)

Operations:

(m,n)	n characters starting with position m of constant string
DS/TR	define storage for rigid temporary
RXD/A(m,n)	real fixed decimal(m,n)
RXB/A(m,n)	real fixed binary(m,n)
ENTRY	define entry point
PROLOG2	prolog code common to all entry points
LABEL(m)	place label number m here
MOVE(m,n)	move information from storage n to storage m, converting type if needed
MULT(r,m,n)	multiply m by n, result to r
PLUS(r,m,n)	add m and n, result to r
RETURN	return from the procedure
ENDBLOCK	terminate text of the procedure

Note

Each storage-defining operation (RXB/A, etc.) advances the storage counter.

Figure 1. An Example of IL2 Code. (concluded)

and according to the rules of PL/I the value of M*N must be determined at the time of reference to S.) The definition of S in IL2 assumes that the pointer to S is in a known location, and this location is also filled in explicitly by IL2 code.

The meaning of an executable operation depends on the types of storage that it references. For instance, there is just one PLUS operator. The kind of PLUS that is intended--whether it is decimal or binary, and what its operand and result precisions are--is determined by the storages for its operands and result. An alternative design would have been to put more of the semantics of an operation into the operation itself, and to have a variety of addition operators. This alternative was rejected because of the variety of type information implicit in an operator; putting that information into the operands seemed to lead to a more uniform structure. The cost of simplifying the operators is that much more information has to be specified for intermediate results.

Operations such as generic selection, extent calculations for storage allocation, and type conversion are implicit in IL1 but are made explicit in IL2. The intent is to do as much of the machine-independent processing as possible before the later passes, and also to allow a single IL2 operation to be translated in a one-to-many manner with as little dependence as possible on the neighboring operations. Some of the executable operations, such as those relating to stream input-output, represent quite complex actions; these operations generally correspond to calls on routines in a run-time library.

The IL2 code is sorted into segments of two kinds: global segments and block segments. The global segments include integer constants, stringed constants (which include some numbers), static storage, virtual storage (for based and overlay-defined variables), and redefinitions (which allow for indirect references); the block segments include rigid automatic storage (for variables of constant size), flexible automatic storage (for variables whose size is known at block entrance), parameters, prologue code, main code, and internal subroutine code. The internal subroutines are compiler-generated code sequences that are needed in more than one place; they do not include user-defined internal procedures.

In PL/I, operators can be applied to arrays as well as to scalars. Since we would like to be able to produce reasonable code for vector machines, it is important to expand operations on arrays in such a way that their nature is apparent. An operation on an array as a whole is therefore represented by an operation on a single element with a symbolic subscript, and the operation is surrounded by array iterators. The iterators are at a sufficiently high level so that transformation into vector instructions is not difficult. Machine dependencies such as stride values are treated by constructing symbol table entries that allow them to be filled in during a later pass.

Since all temporary storages are defined explicitly, the generated IL2 code tends to have lots of storage definition operations. These operations are themselves space-consuming, both because there are so many of them and because of the amount of information needed to specify a data type. With a few identifiable exceptions, a temporary storage can be discarded once it has been used. Thus the symbol table entries for temporary storages need not be kept around indefinitely. References to subscripted elements of arrays do not repeat the type information for the array. Instead, L-type storages are used for subscripted array elements. An L-type storage is defined to contain a pointer, and its defining operation specifies the storage number of the array itself, from which the necessary type information can be derived.

5. PARSING

Parsing in the CIMS PL/I compiler is done using recursive descent (and, implicitly, an LL(1) grammar), operator precedence for expressions, and an escape hatch to handle the places where the language is not LL(1). The parsing actions are represented explicitly in the compiler; that is, the compiler actually contains code to examine tokens one by one and take appropriate action. However, the selection of an appropriate action is automated by means of a function CHOICE. The argument to CHOICE is an arbitrary unique integer that gives the context in which the selection is being made. CHOICE returns an integer that can be used to make the selection, usually as the argument of a computed GO TO (a case-statement, in effect). The value returned by CHOICE is determined by the context, i.e., the argument to CHOICE, and by the next token in the input stream. The call to CHOICE is followed by a comment, written in a standard format, that specifies the value to be returned by CHOICE for various tokens or token classes. The text of the compiler itself is then used as input to a preprocessor that produces the tables needed to implement CHOICE. For example, during the parsing of an assignment-statement we execute the statement

```
        GO TO CASE(CHOICE(18));
```

Following that statement we have the comment

```
        /* CODES FOR CHOICE 18
           1  (
           2  ,
           3  ;
           3  ENDPROC
        */
```

Thus if the next token is a left parenthesis we go to CHOICE(1); if it is a comma we go to CHOICE(2); if it is a semicolon or the end of the procedure we go to CHOICE(3); and if it is none of these we go to CHOICE(0). The trigger for the preprocessor is the sequence of characters "/* CODES FOR CHOICE". The information between this sequence and the following "*/" is just what the preprocessor needs in order to construct the hashed lookup tables used within CHOICE. As an aside, I should mention that these tables are generated in assembly language rather than in PL/I. There are two reasons for this. First, the tables contain pointers, and so they cannot be expressed as PL/I initial attributes. Second, even if they could, some PL/I compilers are reluctant to accept extremely long statements, and table initializations sometimes exceed these compiler limits. (That was my experience with the IBM PL/I F-compiler.)

There are two advantages to the CHOICE mechanism as an aid in implementing a recursive descent parser. First, it relieves the compiler writer of the need to write repetitive code to test alternatives individually -- code that would be inefficient as well as unaesthetic. Second, and more subtly, it keeps all the information about the choice encodings in one place. Since the tables can easily be regenerated, there is no difficulty in keeping them up-to-date. Without some such arrangement, any change to the choice values would require a corresponding manual modification of the parsing tables.

PL/I is not an LL(1) language, and in fact it fails quite badly to fall into any reasonable class of languages on account of the use of keywords as identifiers. This is really more of a problem for the compiler-writer than for the user. In CIMS PL/I an oracle is called whenever we must make a parsing decision for which the LL(1) grammar is inadequate, i.e., a decision for which our one-token lookahead is insufficient. The oracles make use of a lookahead feature of the lexical scanner. The lexical scanner can be instructed to prescan a sequence of tokens, return to its position at the start of the prescan, and scan the same tokens once again (but without repartitioning and reclassifying them). Each oracle looks at as many tokens as necessary in order to determine what syntactic construct is about to be parsed. The actual parse then takes place during a second scan of the tokens. The oracle can ignore the possibility of errors, since any errors will be caught during the actual parse, and an erroneous construct necessarily has no correct classification. Since the oracles are part of the parser, the lexical scanner need not have any knowledge of the higher-level context. The oracles use the same CHOICE mechanism as the rest of the parser.

Statement classification is a good example of how oracles are used. When a statement is about to be parsed, an oracle determines the number of prefixes attached to the statement (either condition prefixes or statement names) and the type of the statement. If the statement starts with a keyword, then it may be necessary to scan quite far in order to determine whether the keyword really is a keyword rather than an identifier that starts an assignment statement. The oracle need not actually parse the statement, however; it only has to extract enough information to classify it, usually looking for pairs of adjacent tokens of certain types. For instance,

```
        IF (2*A) = B THEN
```

is an IF statement, but

```
        IF (2*A) = B ;
```

is an assignment statement. We can tell which is which by looking for THEN preceded by a right parenthesis, identifier, or constant; if we see such a pair before we see a semicolon then we have an IF statement, and otherwise we have an assignment statement. Fortunately this case is much worse than most that are actually encountered, and in fact most decisions can be made by looking ahead only one or two tokens. The whole approach is practical because PL/I is dominantly LL(1), and the oracles account for only a small part of the time needed in parsing. A list of those contexts in which oracles are needed, and the algorithms used by the oracles, are given in Appendix A.

Some explanation is in order as to why an automatic parser, perhaps using an LL(1) grammar, was not used. In fact, Version 1 does use an automatic parser, and that was the cause of a number of problems. Some difficulties with automatic parsers are:

1. The existing ones are all coupled to specific languages used for writing action (i.e., semantic) routines. YACC [3], for instance, can only be used with C or RATFOR. Getting around this limitation would require a significant tool-building effort (which might be justified if not for the other objections).

2. Using automatic parsing, the parser becomes a single monolithic

procedure. There is no obvious way to decouple different parts of the grammar so that they are handled by separately compiled procedures. Moreover, it is difficult to utilize the control flow features of the language in which the action routines are written (recursion in particular) since the parser always remains in control.

3. Communication among the action routines is difficult for a language as complex as PL/I. There is no single type that can appropriately be used for the values returned by the action routines and passed to the next higher-level action routine. (YACC uses integers for this purpose, but that only works because of the choice of C or RATFOR for writing the actions.) Pointers might be used, taking advantage of the fact that in PL/I they are untyped, but that approach would create a different problem: making sure that storage allocated for intermediate results is freed properly. Moreover, the mechanisms for communicating semantic information downwards (from the action routine for a production to the action routines within the production) are weak and artificial.

None of these arguments by itself is conclusive. In fact, developing an automatic parser that meets these objections seems quite feasible and would be well worth doing. The CIMS PL/I parsing method was chosen, not because it was an ideal solution, but because it was a reasonable solution that could be implemented easily, while the alternative -- developing a really satisfactory automatic parser -- was a more speculative enterprise whose difficulty was not so easy to evaluate.

6. ERROR HANDLING

The error-handling facilities were designed according to these criteria:

1. Each pass must be able to generate input that is syntactically acceptable to the next pass.

2. Errors should be corrected only when the correction can be accomplished cheaply, or when the correction is needed in order to produce syntactically valid intermediate forms.

3. Extraneous error complaints should be avoided, even at the cost of not detecting multiple errors.

Good error handling is achieved by generalizing the syntactic forms accepted by the parser, and preparing the later passes to cope with these generalized forms. For example, the parser accepts "F" and "A(3,2,6)" as valid format items, even though neither is acceptable in PL/I. More generally, arbitrary identifiers are accepted in any context where a list of keywords is expected, and a sequence of keywords may contain repetitions or lack necessary elements. The parser will usually issue an error message when it encounters an invalid keyword. It then skips past the keyword and the parenthesized expression-sequence, if any, that follows the keyword.

Each procedure that processes a program unit is obliged to produce a translation that represents some unit of the same kind, even if the translation is only a dim reflection of the original unit. For instance, expressions may contain error objects and declarations may contain an error attribute. An error object or attribute propagates the error property and inhibits complaints about further errors.

Aside from the processing of the error messages and codes themselves by a preprocessor, error handling is not automated. When an unexpected token is found during parsing, a standard syntax error routine is called, and an error flag is set for the statement. Setting the error flag inhibits further error messages about the statement. When the end of the statement is reached, the previous and current token at the point of error are printed. There are similar flags for parenthesis errors. If missing left or right parentheses are found, flags are set accordingly. When the end of the statement is reached, one of the messages "too many right parentheses", "too few right parentheses", or "parenthesis matching error" is printed. (If the error flag is on, these messages will be suppressed.) Thus multiple parenthesis errors within a single statement will produce at most one message.

7. PREPROCESSORS

The tables needed by the CHOICE function are generated by a preprocessor that scans the text of the compiler itself, seeking out lines that match a particular pattern, namely, that they begin with "/* CODES FOR CHOICE". There are also other kinds of information that can usefully be planted in a standard form within the text of the compiler and extracted later by a preprocessor.

The error message generator depends on a preprocessor that scans the compiler looking for the texts of error messages. Each error message has a unique integer code associated with it. During compilation, an error routine is called whenever an error is detected in the source program. The error routine is given an integer code -- the error number -- and possibly a string such as a variable name that is to be substituted into the text of the message itself. In the compiler, the call on the error routine is followed by a comment, again in a standard form, giving the error number, the severity of the error, and the text of the message. An example of such a call and comment is

```
       CALL ERRMESSAGE1(67, LTOKEN_REP);
/* ERROR MESSAGE 67, SEVERITY 3
THE KEYWORD - - IS NOT RECOGNIZED IN THIS CONTEXT
*/
```

Here the current value of LTOKEN_REP (the last token scanned) is substituted for # to generate the appropriate message.

The preprocessor can locate these comments and generate the informa-

ion needed by the routine that actually prints the errors. During a com-
ilation, a list of errors is accumulated. When a procedure has been com-
iled, the errors are sorted by statement number, the message texts are re-
rieved, and strings are substituted into the texts as necessary. The
rror messages are processed by a separate overlay that contains both the
essage texts and the procedures that generate the actual messages sent to
ne user.

A preprocessor is also used to provide a major documentation aid.
ach procedure of the compiler is headed by one or more specifications
ritten in standard form. For each entry point of the procedure, the cor-
esponding specification gives the entry point name, the name and descrip-
ion of each parameter, a description of the returned value, and a narra-
ive statement of what the entry point does. The preprocessor extracts
nese specifications and associates with each entry point the name of the
ource module where the entry point appears. The specifications are then
orted alphabetically by entry point name and printed. The alphabetized
ist of specifications is very helpful in reading the text of the compiler,
ince the reader can look at a portion of the compiler without having to
lip back and forth in order to see what a procedure call does. Moreover,
: is easy to find the module where a particular entry point is defined.

Not all of the preprocessors require the text of the compiler as an
nput. For instance, it is helpful to have a table of bit strings that in-
icate which attributes are in conflict with a given attribute (e.g., FIXED
s in conflict with POINTER). The input for this preprocessor is quite in-
ependent of the text of the compiler.

8. CONCLUDING REMARKS

The main lesson learned from building the CIMS PL/I compiler was that theoretically attractive techniques may fail to work in practice, not because of some single conceptual flaw but rather because of an accumulation of small difficulties. Moreover, constraints from the operating environment may have a much more profound effect on design than one would like; machine-independent designs are far more aesthetically pleasing than machine-dependent ones. The size and complexity of PL/I generate design difficulties that do not occur in smaller and simpler languages -- which could be taken as an argument against PL/I.

This paper highlights only a few of the interesting aspects of CIMS PL/I. Among the subjects not treated here are code generation, design of the interfaces (such as listings) between the system and its users, and the many problems that arise in implementing the run-time library. The run-time library is the great hidden pitfall of compiler writing; few books discuss it, but yet the work needed in order to create it is often of the same order of magnitude as the work needed to build the compiler itself.

References

American National Standards Institute, "Programming Language PL/I". Document ANSI X3.53-1976, New York.

Freiburghouse, R., The Multics PL/I Compiler. Fall Joint Computer Conference, 1969, pp. 187-199.

Johnson, S., YACC - Yet Another Compiler-Compiler. Computing Science Tech. Rpt. 32, Bell Telephone Labs., Murray Hill, NJ, 1975.

APPENDIX A
CONTEXTS WHERE ORACLES ARE NEEDED

In this appendix we give examples of those constructs in PL/I that cannot be distinguished using an LL(1) grammar, and indicate how those constructs are distinguished in CIMS PL/I. In the examples, bal indicates a parenthesis-balanced sequence of symbols. As our only objective is to distinguish the constructs from each other, we don't need to analyze their internal structure or validate them; any errors will be caught during the actual parse. For each construct, we give examples of sequences that are difficult to distinguish; indicate the syntactic category of each sequence; and give the method used to distinguish the sequences.

1.
```
ON ERROR SYSTEM   ;  /* SYSTEM IS A KEYWORD */
ON ERROR SYSTEM=3;  /* SYSTEM=3 IS AN ASSIGNMENT */
```

If SYSTEM is followed by ";" it is a keyword; otherwise not.

2.
```
GET:               /* GET IS A STATEMENT-NAME */
GET DATA;          /* GET-STATEMENT */
GET->X=5;          /* ASSIGNMENT-STATEMENT */
GET.X=5;           /* ASSIGNMENT-STATEMENT */
GET=5;             /* ASSIGNMENT-STATEMENT */
GET,X=5;           /* ASSIGNMENT-STATEMENT */
STOP;              /* STOP-STATEMENT */
GET(3):            /* GET IS A STATEMENT-NAME */
GET(3)->X=5;       /* ASSIGNMENT-STATEMENT */
GET(3).X=5;        /* ASSIGNMENT-STATEMENT */
GET(3)=5;          /* ASSIGNMENT-STATEMENT */
GET(3),X=5;        /* ASSIGNMENT-STATEMENT */
STOP(3):           /* STOP IS A STATEMENT-NAME */
```

In order to distinguish statement keywords, statement-names, and identifiers in any context other than "IF(bal)=" or "DECLARE(bal),", examine the token T that follows the initial token of the statement. If T is anything other than "(", the category of T determines whether or not the initial token is a keyword, a statement-name, or an identifier. If T is "(", then find the matching ")" and make the determination in the same way as after the initial token.

3.
```
IF(5)=THEN+THEN;          /* ASSIGNMENT-STATEMENT */
IF(5)=THEN+THEN THEN;     /* IF-STATEMENT */
```

In order to distinguish an if-statement from an assignment-statement when the statement begins with "IF(bal)=", look for ";", or for "THEN" not preceded by an operator, ".", ",", or "(". If ";" is found first then the statement is an assignment-statement; otherwise it is an if-statement.

4.
```
DECLARE(A,B),C(3)=CHAR(12);    /* ASSIGNMENT-STATEMENT */
DECLARE(A,B),C(3) CHAR(12);    /* DECLARE-STATEMENT */
DECLARE(A,B),C(3)         ;    /* DECLARE-STATEMENT */
```

In order to distinguish a declare-statement from an assignment-statement when the statement begins with "DCL(bal),", look for ";" or for an integer, identifier, or ")" followed by an identifier or ";". If we see ";" first (and it is not preceded by an integer, identifier, or ")") then the statement is an assignment-statement; otherwise it is a declare-statement.

5.
```
ELSE(SIZE):X=3;           /* (SIZE) IS A CONDITION-PREFIX */
ELSE(SIZE)  =3;           /* ASSIGNMENT TO ELSE(SIZE) */
ELSE(2)     =3;           /* ASSIGNMENT TO ELSE(2) */
ELSE(2):   X=3;           /* ELSE(2) IS A STATEMENT-NAME */
ELSE:      X=3;           /* ELSE IS A STATEMENT-NAME */
ELSE STOP;                /* ELSE IS A KEYWORD */
ELSE ;                    /* ELSE IS A KEYWORD */
ON ERROR SNAP  ;          /* SNAP IS A KEYWORD */
ON ERROR SNAP=5;          /* ASSIGNMENT TO SNAP */
```

"SNAP" following an on-statement behaves like "ELSE". If "ELSE" is followed by ":", then the "ELSE" is a statement-name. If "ELSE" is followed by a token other than "(" or ":", then the "ELSE" is a keyword. If the "ELSE" is followed by "(", then classify "ELSE" according to (1) whether the material up to the matching ")" contains an identifier, and (2) whether the token after the matching ")" is ":".

6.
```
DO WHILE(P=0);          /* DO WHILE */
DO WHILE(P)=0;          /* DO WITH CONTROL VARIABLE WHILE(P) */
```

Find the ")" matching the "(" after "WHILE". If the next token is ";" the statement is a do-while; otherwise the statement is a do-statement with control variable.

7.
```
PUT LIST((A(I) DO I=1 TO 10));  /* DO WITH ITERATED LIST */
PUT LIST((A(I))            );  /* LIST WITH SINGLE ITEM (A(I))
*/
```

To determine whether a "(" in a data-list starts a parenthesized item or an iteration-list, look for "DO" not preceded by ".", ",", "(", or an operator, and not within nested parentheses. If such a "DO" is found before the matching ")" is found, the original "(" starts an iteration-list, and otherwise it is part of a parenthesized item.

8.
```
DCL A(50) INITIAL ((N+6),12);    /* (N+6) IS A SINGLE ITEM */
DCL A(50) INITIAL ((N+6));       /* (N+6) IS A SINGLE ITEM */
DCL A(50) INITIAL ((N+6)3);      /* N+6 ITERATES A SINGLE ITEM */
DCL A(50) INITIAL ((N+6)-3);     /* N+6 ITERATES A SINGLE ITEM */
DCL A(50) INITIAL ((N+6)*);      /* N+6 ITERATES A SINGLE ITEM */
DCL A(50) INITIAL ((N+6)J);      /* N+6 ITERATES A SINGLE ITEM */
DCL A(50) INITIAL ((6)'A');      /* 6 IS A STRING REPETITION
FACTOR */
DCL A(50) INITIAL ((6)(3)'A');   /* 6 ITERATES A SINGLE ITEM */
DCL A(50) INITIAL ((6)(3,'A')); /* 6 ITERATES A LIST */
```

To classify the "(" at the beginning of an "INITIAL" list, find the matching ")" and look at the following token T. If T is anything other than "(", we can classify immediately. If T is "(", then see if T is followed immediately by the sequence integer, ")", string. If so, the first "(" introduces iteration over a single item; otherwise it introduces iteration over a list.

NYU
Comp.Sci.Dept.
TR-013 c.2

Abrahams
The CIMS PL/I compiler.

NYU
Comp.Sci.Dept.
TR-013 c.2
Abrahams

AUTHOR	
The CIMS PL/I compiler.	
TITLE	

DATE DUE	BORROWER'S NAME
	[illegible]

NOV [illegible]			